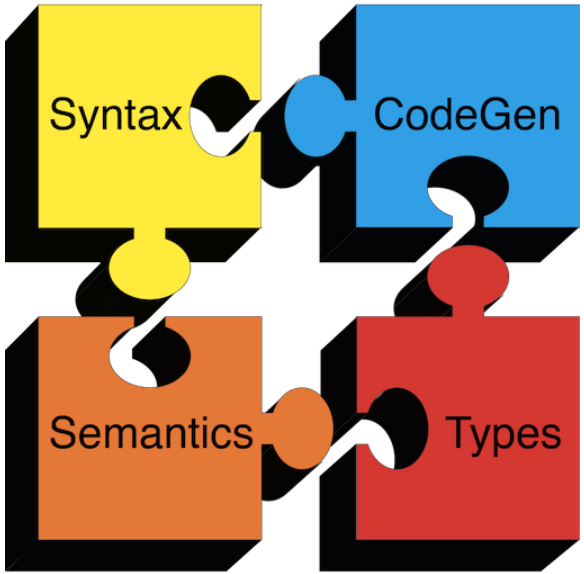

Compilers

Cool Semantics II

Alex Aiken

Cool Semantics II

- Informal semantics of new T
  - Allocate locations to hold all attributes of an object of class T
    - Essentially, allocate a new object
  - Set attributes with their default values
  - Evaluate the initializers and set the resulting attribute values
  - Return the newly allocated object

Cool Semantics II

- For each class A there is a default value D_A
  - D_{int} = Int(0)
  - D_{bool} = Bool(false)
  - D_{string} = String(0, “”)
  - D_A = void (for any other class A)

Cool Semantics II

- For a class A we write

class(A) = ($a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n$) where

  - a_i are the attributes (including the inherited ones)
  - T_i are the attributes' declared types
  - e_i are the initializers

Cool Semantics II

T_0 = if (T == SELF_TYPE and so = X(…)) then X else T
class(T_0) = (a_1 : T_1 ← e_1,…, a_n : T_n ← e_n)
l_i = newloc(S) for i = 1,…,n
v = T_0(a_1= l_1,…,a_n= l_n)
S_1 = S[D_{T1}/l_1,…,D_{Tn}/l_n]
E' = [a_1 : l_1, …, a_n : l_n]
v, E', S_1 ⊢ { a_1 ← e_1; …; a_n ← e_n; } : v_n, S_2

---

so, E, S ⊢ new T : v, S_2

Cool Semantics II

- The first three steps allocate the object
- The remaining steps initialize it
  - By evaluating a sequence of assignments
- State in which the initializers are evaluated
  - Self is the current object
  - Only the attributes are in scope (same as in typing)
  - Initial values of attributes are the defaults

Cool Semantics II

- Informal semantics of $e_0.f(e_1,\ldots,e_n)$
  - Evaluate the arguments in order $e_1,\ldots,e_n$
  - Evaluate e_0 to the target object
  - Let X be the dynamic type of the target object
  - Fetch from X the definition of f (with n args.)
  - Create n new locations and an environment that maps f's formal arguments to those locations
  - Initialize the locations with the actual arguments
  - Set self to the target object and evaluate f's body

Cool Semantics II

- For a class A and a method f of A (possibly inherited):

impl(A, f) = (x_1, …, x_n, e_{body}) where

- x_i are the names of the formal arguments
- e_{body} is the body of the method

Cool Semantics II

$$\frac{\begin{array}{l} so, E, S \vdash e_1 : v_1, S_1 \\ so, E, S_1 \vdash e_2 : v_2, S_2 \\ \ldots \\ so, E, S_{n-1} \vdash e_n : v_n, S_n \\ so, E, S_n \vdash e_0 : v_0, S_{n+1} \\ v_0 = X(a_1 = l_1, \ldots, a_m = l_m) \\ impl(X, f) = (x_1, \ldots, x_n, e_{body}) \\ l_{xi} = newloc(S_{n+1}) \text{ for } i = 1, \ldots, n \\ E' = [a_1 : l_1, \ldots, a_m : l_m][x_1/l_{x1}, \ldots, x_n/l_{xn}] \\ S_{n+2} = S_{n+1}[v_1/l_{x1}, \ldots, v_n/l_{xn}] \\ v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3} \end{array}}{so, E, S \vdash e_0.f(e_1, \ldots, e_n) : v, S_{n+3}}$$

Cool Semantics II

What is the final value of S_5 in the dispatch of obj.foo(i) below?

```
Class C {
        a: Int <- 0;
        foo(x: Int) : Int { x + a };
};
```

so, $[i:l_i]$, $S_1 \vdash i : 3, S_2$
so, $[i:l_i]$, $S_2 \vdash obj : C(a = l_{obj_a}), S_3$
$impl(C, foo) = (x, x + a)$
$l_x = newloc(S_3)$
$S_4 = S_3[3/l_x]$

$$\frac{C(a = l_{obj_a}), [a:l_{obj_a}][x/l_x], S_4 \vdash x + a : 4, S_5}{so, [i:l_i], [l_{obj_a} \leftarrow 1, l_i \leftarrow 3] \vdash obj.foo(i) : 4, S_5}$$

- ○ $[l_i \leftarrow 3]$
- ○ $[l_{obj_a} \leftarrow 1, l_i \leftarrow 3]$
- ○ $[l_{obj_a} \leftarrow 1, l_i \leftarrow 3, l_x \leftarrow 3]$
- ○ It cannot be determined from the information given.

Cool Semantics II

- The body of the method is invoked with
  - E mapping formal arguments and self's attributes
  - S like the caller's except with actual arguments bound to the locations allocated for formals
- The notion of the frame is implicit
  - New locations are allocated for actual arguments
- The semantics of static dispatch is similar

Runtime Errors

Operational rules do not cover all cases

Consider the dispatch example:

$$
\begin{array}{l}
\ldots \\
so, E, S_n \vdash e_0 \ : v_0, S_{n+1} \\
v_0 = X(a_1 = l_1, \ldots, a_m = l_m) \\
impl(X, f) = (x_1, \ldots, x_n, e_{body}) \\
\ldots
\end{array}
$$

Cool Semantics II

- There are some runtime errors that the type checker does not prevent
  - A dispatch on void
  - Division by zero
  - Substring out of range
  - Heap overflow

- In such cases execution must abort gracefully
  - With an error message, not with a segfault

Cool Semantics II

- Operational rules are very precise & detailed
  - Nothing is left unspecified
  - Read them carefully
- Most languages do not have a well specified operational semantics
- When portability is important an operational semantics becomes essential